

---

# Python 對 Linux perf 分析器的支援

發 3.14.0a1

Guido van Rossum and the Python development team

11 月 11, 2024

Python Software Foundation  
Email: docs@python.org

## Contents

1	如何 用 perf 分析支援	3
2	如何獲得最佳結果	4
3	How to work without frame pointers	4
	索引	6

---

### 作者

Pablo Galindo

Linux 性能分析器 (Linux perf profiler) 是一個非常大的工具，可讓你分析 獲取有關應用程式的性能資訊。perf 還擁有一個非常活躍的工具生態系統，有助於分析其生成的資料。

在 Python 應用程式中使用 perf 分析器的主要問題是 perf 僅獲取有關原生符號的資訊，即用 C 編寫的函式和程式的名稱。這表示程式碼中的 Python 函式名稱和檔案名稱不會出現在 perf 的輸出中。

從 Python 3.12 開始，直譯器可以在特殊模式下執行，該模式允許 Python 函式出現在 perf 分析器的輸出中。 用此模式後，直譯器將在執行每個 Python 函式之前插入 (interpose) 一小段動態編譯的程式碼， 使用 perf map 檔案來告訴 perf 這段程式碼與相關聯的 Python 函式間的關係。

### 備

目前對 perf 分析器的支援僅適用於 Linux 的特定架構上。檢查 configure 建構步驟的輸出或檢查 `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` 的輸出來查看你的系統是否支援。

例如，參考以下 本：

```
def foo(n):
    result = 0
    for _ in range(n):
        result += 1
    return result

def bar(n):
    foo(n)
```

(繼續下一頁)

(繼續上一頁)

```
def baz(n):
    bar(n)

if __name__ == "__main__":
    baz(1000000)
```

我們可以執行 perf 以 9999 赫[F]樣 CPU 堆[F]追[F] (stack trace):

```
$ perf record -F 9999 -g -o perf.data python my_script.py
```

然後我們可以使用 `perf report` 來分析資料：

```
$ perf report --stdio -n -g

# Children      Self          Samples  Command      Shared Object      Symbol
# .....
↪ .....
#
    91.08%      0.00%              0  python.exe  python.exe        [.] _start
    |
    |--_start
    |
    |--90.71%--__libc_start_main
    |         Py_BytesMain
    |         |
    |         |--56.88%--pymain_run_python.constprop.0
    |         |
    |         |         |--56.13%--_PyRun_AnyFileObject
    |         |         |         _PyRun_SimpleFileObject
    |         |         |
    |         |         |         |--55.02%--run_mod
    |         |         |         |
    |         |         |         |         |--54.65%--PyEval_EvalCode
    |         |         |         |         |         _PyEval_EvalFrameDefault
    |         |         |         |         |         PyObject_Vectorcall
    |         |         |         |         |         _PyEval_Vector
    |         |         |         |         |         _PyEval_EvalFrameDefault
    |         |         |         |         |         PyObject_Vectorcall
    |         |         |         |         |         _PyEval_Vector
    |         |         |         |         |         _PyEval_EvalFrameDefault
    |         |         |         |         |         PyObject_Vectorcall
    |         |         |         |         |         _PyEval_Vector
    |         |         |         |         |         |         |--51.67%--_PyEval_
↪ EvalFrameDefault
    |         |         |         |         |         |
    |         |         |         |         |         |         |--11.52%--_
↪ PyLong_Add
    |         |         |         |         |         |
    |         |         |         |         |         |         |--
↪ 2.97%--_PyObject_Malloc
    ...
```

如你所見，Python 函式未顯示在輸出中，僅顯示 `_Py_Eval_EvalFrameDefault`（[F](#) Python 位元組碼 (bytecode) 求值的函式）。不幸的是，這不是很有用，因[F](#)所有 Python 函式都使用相同的 C 函式來替位元組碼求值，因此我們無法知道哪個 Python 函式是對應於哪個位元組碼計算函式。

作 $\mathbb{F}$ 替代，如果我們在 $\mathbb{F}$ 用 `perf` 支援的情 $\mathbb{F}$ 下執行相同的實驗，我們會得到：

```
$ perf report --stdio -n -g
```

(繼續下一頁)

# Children	Self	Samples	Command	Shared Object	Symbol		
# .....	.....	.....	.....	.....	.....		
↪.....							
#							
90.58%	0.36%	1	python.exe	python.exe	[.] _start		
	---_start						
	--89.86%--	__libc_start_main					
		Py_BytesMain					
		--55.43%--	pymain_run_python.constprop.0				
			--54.71%--	_PyRun_AnyFileObject			
				_PyRun_SimpleFileObject			
				--53.62%--	run_mod		
				--53.26%--	PyEval_EvalCode		
↪py					py::<module>:/src/script.		
					_PyEval_EvalFrameDefault		
					PyObject_Vectorcall		
					_PyEval_Vector		
					py::baz:/src/script.py		
					_PyEval_EvalFrameDefault		
					PyObject_Vectorcall		
					_PyEval_Vector		
					py::bar:/src/script.py		
					_PyEval_EvalFrameDefault		
					PyObject_Vectorcall		
					_PyEval_Vector		
					py::foo:/src/script.py		
					--51.81%--	_PyEval_	
↪EvalFrameDefault							
						--13.77%--	_
↪PyLong_Add							
↪3.26%--	_PyObject_Malloc						--

## 1 如何`☐`用 perf 分析支援

要`☐`用 perf 分析支援，可以在一開始就使用環境變數 `PYTHONPERFSUPPORT` 或使用 `-X perf` 選項，也可以使用 `sys.activate_stack_trampoline()` 和 `sys.deactivate_stack_trampoline()` 來動態`☐`用。

`sys` 函式優先於 `-X` 選項，`-X` 選項優先於環境變數。

例如，使用環境變數：

```
$ PYTHONPERFSUPPORT=1 perf record -F 9999 -g -o perf.data python script.py
$ perf report -g -i perf.data
```

例如，使用 `-X` 選項：

```
$ perf record -F 9999 -g -o perf.data python -X perf script.py
$ perf report -g -i perf.data
```

例如，在 `example.py` 檔案中使用 `sys` API：

```
import sys

sys.activate_stack_trampoline("perf")
do_profiled_stuff()
sys.deactivate_stack_trampoline()

non_profiled_stuff()
```

... 然後：

```
$ perf record -F 9999 -g -o perf.data python ./example.py
$ perf report -g -i perf.data
```

## 2 如何獲得最佳結果

為了獲得最佳結果，應使用 `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"` 來進行 Python 編譯，因為這能允許分析器僅使用 `frame` 指標而不是 `DWARF` 除錯資訊來解析 (`unwind`)。這是因為，由於插入以允許 `perf` 支援的程式碼是動態生成的，因此它沒有任何可用的 `DWARF` 除錯資訊。

你可以透過執行以下指令來檢查你的系統是否已使用此旗標進行編譯：

```
$ python -m sysconfig | grep 'no-omit-frame-pointer'
```

如果你有看到任何輸出，則表示你的直譯器尚未使用 `frame` 指標進行編譯，因此它可能無法在 `perf` 的輸出中顯示 Python 函式。

## 3 How to work without frame pointers

If you are working with a Python interpreter that has been compiled without frame pointers, you can still use the `perf` profiler, but the overhead will be a bit higher because Python needs to generate unwinding information for every Python function call on the fly. Additionally, `perf` will take more time to process the data because it will need to use the `DWARF` debugging information to unwind the stack and this is a slow process.

To enable this mode, you can use the environment variable `PYTHON_PERF_JIT_SUPPORT` or the `-X perf_jit` option, which will enable the JIT mode for the `perf` profiler.

### 備註

Due to a bug in the `perf` tool, only `perf` versions higher than v6.8 will work with the JIT mode. The fix was also backported to the v6.7.2 version of the tool.

Note that when checking the version of the `perf` tool (which can be done by running `perf version`) you must take into account that some distros add some custom version numbers including a `-` character. This means that `perf 6.7-3` is not necessarily `perf 6.7.3`.

When using the `perf` JIT mode, you need an extra step before you can run `perf report`. You need to call the `perf inject` command to inject the JIT information into the `perf.data` file.:

```
$ perf record -F 9999 -g --call-graph dwarf -o perf.data python -Xperf_jit my_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

或使用環境變數：

```
$ PYTHON_PERF_JIT_SUPPORT=1 perf record -F 9999 -g --call-graph dwarf -o perf.data python my_
↳script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

`perf inject --jit` command will read `perf.data`, automatically pick up the `perf` dump file that Python creates (in `/tmp/perf-$PID.dump`), and then create `perf.jit.data` which merges all the JIT information together. It should also create a lot of `jitted-XXXX-N.so` files in the current directory which are ELF images for all the JIT trampolines that were created by Python.

#### 警告

Notice that when using `--call-graph dwarf` the `perf` tool will take snapshots of the stack of the process being profiled and save the information in the `perf.data` file. By default the size of the stack dump is 8192 bytes but the user can change the size by passing the size after comma like `--call-graph dwarf,4096`. The size of the stack dump is important because if the size is too small `perf` will not be able to unwind the stack and the output will be incomplete. On the other hand, if the size is too big, then `perf` won't be able to sample the process as frequently as it would like as the overhead will be higher.

## 索引

### 非依字母順序

環境變數

`PYTHON_PERF_JIT_SUPPORT`, 4

`PYTHONPERFSUPPORT`, 3

### P

`PYTHON_PERF_JIT_SUPPORT`, 4

`PYTHONPERFSUPPORT`, 3